Lambdas and Functional Programming

18

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Functional programming.
- Higher-order and first-class functions.
- Pure functions and their characteristics.
- Recursion functions, evaluation, and referential transparency.
- Working examples of all functions using Java.
- Lambdas.

18.1 | Introduction

Programming applications in different languages require a specific style known as programming paradigm. In the earlier days, programmers used to code in C via procedural programming. Procedural programming is also called *imperative programming*. In a procedural paradigm, the code runs step-by-step, similar to kitchen recipes. The paradigm was excellent for its time, but as the complexities in the developing world grew, it was realized that there was a need for a reusable code that could make programming easier for all the stages of software lifecycle.

Companies revolutionized their business processes by adding a desktop or web system to achieve greater business productivity. Programmers soon realized that procedural programming was ineffective and cumbersome for their requirements.

As a result, object-oriented paradigm was introduced. In object-oriented programming (OOP), "objects" were introduced that mirrored the real world where each object can have its distinct behavior and state. Thus, developers began viewing code with respect to the real world and added functionalities accordingly.

OOP was globally instrumental in shrinking the size of codebases. As users were able to assign behavior and states to each object, they used OOP components such as inheritance, polymorphism, encapsulation, and abstract to achieve unforeseen levels of productivity. The programs that once required 5,000 lines of code were now reduced to only 500 lines.

OOP manages to resolve several issues, but with advancements in academic as well as industry landscapes, it was simply not enough to work in all types of environments. Thus, a newer programming paradigm achieved success and became popular for programming. This new paradigm, called *functional programming*, is supported by mainstream languages such as Python, PHP, and Java, while several languages such as Haskell are gaining prominence due to their effectiveness for functional-based applications.

18.2 | Functional Programming

As the name suggests, functional programming relies on "functions". Here, functions can refer to any operation or feature that has to be added into the application. With functional programming, there is no requirement for modifying the other components of the code in the process of adding a new function.

These functions can be seen as similar to real-world formulas in mathematics. In fact, functional programming is heavily based on mathematical functions. In mathematics, a problem is tackled by reasoning and solving with a technique that the real theory behind all the complex steps of the function are abstracted, and the problem solver can focus more on higher-level complexities by designing a solution. Likewise, functional programming can be seen as providing a greater level of abstraction in software development.

Functional programming prevents modification in states and mutability of data. Instead of statements, expressions are used in functional programming.

How is functional programming different from object-oriented programming?

18.2.1 Fundamental Concepts of Functional Programming

Before writing functional paradigm-based code, it is necessary to familiarize ourselves with the following concepts for the production of quality code. Without properly interpreting the underlying theory behind functional programming, our implementation can suffer in the production of powerful code.

18.2.1.1 Higher-Order and First-Class Functions

A higher-order function exists in both mathematical and computer science worlds. Generally, a higher-order function can be described as possessing any of the two properties.

1. The result type of the function is also a function. For instance:

```
// Here A is a function that takes an integer value of total
function A ( int total)
{
    // The function performs some processing and adds it to return another function
return marks() ;
}
```

2. The function's arguments include a single function or multiple functions. For example, we have defined a function `areaofRectangle`. The function takes the argument in the form of another function `values` that saves length and breadth of the rectangle and passes it to the former function.

```
// Here areaofRectangle is a function that has taken another function values for the
calculation of area
public areaofRectangle (values)
{
    // Since values is returned so the function can be considered as a higher-order
function.
    return values;
}
```

Other than higher-order functions, we also have to acquaint ourselves with first-class functions. First-class functions are similar to higher-order functions. However, in first-class function, it is mandatory to follow both the conditions of a higher-order function; that is, a first-class function must return another function and contain a function for its parameter arguments also. Thus, each first-class function can be termed as higher-order function.

However, there is a thin line that separates both types of functions. When we use the term "higher order", our focus is more inclined towards the mathematical aspect of the problem, while "first class" is used to view the problem in a computer science oriented approach.

18.2.1.2 Pure Functions

As already discussed, one of the core components of a functional application is the presence of expressions. These expressions are also called *pure functions*. They are beneficial because they do not incur any significant side effects on input/output operations or memory. In functional programming, a function has a side effect when it is able to change states of data that lie outside its scope. A pure function has the following characteristics:

1. Sometimes, an expression is useful for a while. However, with the passage of time newer changes to the application may render the expression as useless. In other paradigms where statements are used in place of expressions, removing these statements may have an adverse impact on the application. However, in functional programming by eliminating expressions from an application, there is no effect on the operation of any other expression.

2. In computer science, sometimes a certain strategy called *memoisation* is used for increasing the speed of computer programs. In memoisation, the results of a function are stored in the cache so they can be returned when the function is called using the same inputs. Pure functions support memoisation; that is, if a pure function uses the same inputs as arguments and is continuously called by the applications, the system resources can be saved by issuing the same result through caching.
3. When side effects are not supported by the complete language, then a compiler can process expressions by any evaluation technique.
4. Pure functions can employ parallelism. What this means is that if the data that is stored by two pure expressions is separate from any relationship (logical or data dependency), then both expressions can be simultaneously processed. Likewise, they can also be rearranged.

QUICK CHALLENGE Give an example of pure function.

18.2.1.3 Recursion

In procedural and objective paradigms, looping (iteration) of elements is performed with the use of loops (for loop, while loop). On the other hand, functional languages are geared towards the use of recursion for looping.

Recursion is a programming concept in which a function calls (invokes) itself. A recursive function contains conditions that repeat calling the function until it reaches to a base case. Some people believe that using loops or recursion is a matter of preference. However, recursion is productive in many cases where it:

1. Reduces the lines of code.
2. Reduces the possibility of errors.
3. Reduces the cost of the function.

Recursion techniques can be used by higher-order functions by utilizing anamorphisms and catamorphisms. As a result, the higher-order functions contribute in the development of control structures (like loops) in procedural programming languages.

Usually, functional languages support unrestricted recursions and are also equipped with Turing complete. Thus, the halting problem is undecidable; that is, an issue cannot be responded with a yes or no conclusion. A halting problem is one in which there is an issue of determining whether a computer program (containing an input) has to be stopped or allowed to run for an infinite period of time.

Recursion in functional programming paradigms also need some sort of "inconsistency" that is added into the logic of the language's *type system*. (The set of rules that adds the type property to a construct in a programming language is called type system.) A few functional languages limit recursion types and may allow only the use of well-founded recursion.

QUICK CHALLENGE Define Anamorphisms and Catamorphisms.

18.2.1.4 Evaluation

A function language can be grouped by two factors – strict and non-strict. These factors are influenced by the language's ability to evaluate an expression with the arguments of the function. To understand strict evaluation and non-strict evaluation, let us take the following example where we have to print length of 5 elements:

```
print length([2-1, 5+7, 1/0, 8*8, 4+9])
```

If this expression gets evaluated by a non-strict evaluation, then a length of 5 is returned (as there are 5 elements in the list). Non-strict evaluation does not attempt to go into the intricacies of the elements. On the other hand, if strict evaluation is used, then the expression does not succeed because it finds an error in the third element of the list (1/0). Thus, strict evaluation delves into the details of expression to determine the logic and semantics of the elements. However, when the elements of a function are required for evaluation due to some calling, then non-strict evaluation assesses them in the same manner as strict evaluation.

Non-strict evaluation is implemented using graph reduction. Languages such as Haskell and Clean adopt non-strict evaluation.

18.2.1.5 Referential Transparency

Assignments statements are not supported in a functional program. What this means is that when a variable is assigned and defined with a value, then it cannot be modified in the future. Due to this property, the side effects of an application are reduced to a significant extent as the variables' values are changeable in the duration of execution processes. Hence, functional programs can be considered as referentially transparent.

Suppose there is a basic program in C that has the following expression, where the value of *a* keeps getting changed because of successive evaluations:

```
a = a * 20
```

Let us consider the initial value of *a* as 1. Now, after a single evaluation the value of *a* becomes 20. An additional evaluation can increase the value of *a* to 400. Since using any of these values changes the program's semantics, the expression cannot be said as referentially transparent. This is because its actual value is being continuously changed implicitly. In functional programming, the above example can be changed into the following function:

```
int addonce(int a) {return a+2;}
```

Now, this evaluation always returns a fixed answer as the value of *a* is not modified implicitly. Likewise, the function does not incur with any side effect. Hence, it can be said that functional programs are inherently referentially transparent.

QUICK CHALLENGE Give an example of referential transparency.

18.3 | Functional Programming in Java

Due to the growing demand of the functional paradigm in development circles, the release of Java 8 and Java 9 came with extensive support for functional programming. While functional programming was still possible in earlier versions of Java, there were too many restrictions to take Java seriously. However, with Java 8 and the recent Java 9 release, Oracle has transformed Java as a formidable option for functional enthusiasts. Let us visit some of the functional aspects in Java.

Flash Quiz What are the other languages that support functional programming? What advantage does Java have over them?

18.3.1 Pure Function

Earlier we talked about pure function as a general concept. In Java 9, a pure function can be written as:

```
public class PureFnObject{
// the values of x and y are taken as input arguments
    public int subtract(int x, int y) {
        // The function returns the subtracted answer
    return x - y;
}
}
```

There are two factors that make this function a pure one. First, you can observe that the subtract function is entirely reliant on the input arguments. Second, there are no side effects of the function because the values of *x* and *y* can be only changed inside the function. On the other hand, the same example can be written in the following code where the function is non-pure.

```
public class NonpureFnObject{
    private int x = 0;
    public int subtract(int x) {
        this.x -= x;
        return this.x;
    }
}
```

In this example, the value of the variable *x* is calculated by using the member variable, which means it can be easily modified. Thus, it is changeable and the function can be said to be carrying a side effect.

QUICK CHALLENGE Create a chart to show the difference between pure and non-pure functions.

18.3.2 Higher-Order Function

In Java, a higher-order function can be replicated by adding a single or multiple lambda expressions in the parameters of a function. Additionally, the function must return a lambda expression. Let us see the following example.

```
public class HigherOrderFn {
    public <T> IFactory<T> createFactory(IProducer<T> prod, IConfigurator<T> conf){
        // the lambda expression begins
        return () -> { // the arrow represents the lambda expression
            T ins = prod.produce ();
            conf.configure (ins);
            return ins;
        } // the lambda expression ends
    }
}
```

Observe that a lambda expression is returned by method of `createFactory()`. For this reason, the *t* condition of a higher-order function is verified.

18.3.3 No State

There are no states in a functional program. Here, state means that the member cannot point (reference) to a variable of the object or class. Thus, it can only reference its own variables. For instance, a function with an external state is given as follows:

```
public class Subtract {
    private int beginningValue = 2;
    public int sub(int x) {
        // the original value is modified by addition
    return beginningValue + x;
    }
}
```

Now, observe another example where a function does not have an external state. Here, the values of the function cannot be changed.

```
public class Subtract {
    public int sub(int x, int y) {
        return x + y;
    }
}
```

18.3.4 Functional Interfaces

In Java, an interface with a single abstract method is called functional interface. As you know, abstract methods are those that are incomplete and are not implemented. Interfaces can have both static and default methods (implemented). A functional interface must have a single method which is not implemented. For example:

```
public interface TestingInterface {
    public void run();
}
```

Let's observe the following program:

```
public interface TestingInterface2 {
    public void run();
    public default void hello() {
        System.out.println("hello world");
    }
    public static void helloStatic() {
        System.out.println("static hello world");
    }
}
```

You must be thinking that since two methods that have been implemented, the interface may not be functional. But since the run method is abstract (i.e., it has not been implemented), our interface is still functional. Interfaces that have multiple abstract methods cannot be categorized as a functional interface.

18.4 | Object-Oriented versus Functional Programming

So far you have learnt object-oriented and functional programming. Now let us do the head-to-head comparison between them. Table 18.1 compares these two in multiple areas.

Table 18.1 Object-oriented and functional programming comparison

Parameter	Functional Programming	Object-Oriented Programming
Focus	It focuses on evaluation of functions	It focuses on objects
Data	It mainly uses immutable data	It mainly uses mutable data
Model	It follows declarative programming model	It follows imperative programming model
Parallel Programming	It supports parallel programming	It does not support parallel programming
Execution	Statements executed in any order	Statements executed in a specific order
Iteration	It uses recursion	It uses loops
Elements	Variables and functions	Objects and methods
Use	Used when there are a few things with large number of operations	Used when there are large number of things with a few operations
State	It does not care about any state	It does care about the state

(*Continued*)

Table 18.1 (*Continued*)

Parameter	Functional Programming	Object-Oriented Programming
Primary Unit	Function	Object
Abstraction Support	It supports abstraction over data and behavior	It supports abstraction over data only
Performance on Big Data Processing	Very good	Not so good
Conditional Statements	No support	Supports statements like if-else, switch, etc.
Example	employees.stream().filter(emp -> emp.salary < minSalary).forEach(emp -> System.out.println(emp));	for (Employee emp: employees) { if (emp.getSalary() < minSalary) { System.out.println(emp); } }

How is Stream different from Collection?

18.5 | Lambdas

Now that you have understood the concept of functional interfaces, let us proceed to lambdas. When Java 8 was released, it gained significant traction with the introduction of lambdas. So what exactly are lambdas expressions?

Lambda expressions are an attempt by Java to enhance functional programming. A lambda expression assists a single or multiple instances of a functional interface via concrete implementations. Since lambdas are capable for concrete implementation of a single function, they are appropriate for functional interfaces. They can be created without necessarily adding them to a class. Lambda expressions can be treated as an object and can be passed and executed like an object. The format of a lambda expression is as follows:

```
parameter -> the body of expression
```

Here, the arrow operation is used to represent it.

Is using lambda faster than traditional Java programming?

18.5.1 Elements of Lambda Expression

Following are the main elements you need to know in order to code in Lambda.

18.5.1.1 Parameters

In a typical lambda expression which consists of multiple parts and each part is connected via an arrow. We will learn more about the meaning of these parts in the subsequent sections. On the left side, we can have *n* number of parameters, where *n* can also be a 0. Adding the type of parameter is optional. The type of parameter can be determined by the compiler by sifting through the coding context. Multiple parameters are entailed in round brackets, (). For single parameters, the use of a round brackets is optional. If there is no parameter, then it can be represented by an empty round bracket.

18.5.1.2 Body

Similar to the parameters, the body may also carry *n* number of statements. These statements are represented by curly brackets, {}. However, a single statement does not require curly brackets. The body expression also embodies the return type of the function.

Before going into lambdas, let us check the following example:

```
package java11.fundamentals.chapter18;
public class SimpleMethodExample {

    public void printDemo(String simpleText) {
        // A method to print the String
        System.out.println(simpleText);
    }
    public static void main(String[] args) {
        // Creating an instance of the object for our LamdaExample class
        SimpleMethodExample smEx = new SimpleMethodExample();

        // Assign a value to the object's string
        String simpleText = "A Simple String";
        smEx.printDemo(simpleText);
    }
}
```

The above program produces the following result.

```
A Simple String
```

You may have found the above example as a traditional approach where the implementation of the method is kept hidden from the caller. Here, the caller takes a variable which is then passed to a method `printDemo`. As you can see, there is a side effect issue.

Similarly, let us visit another example in which we are passing a behavior instead of a variable. We have utilized a functional interface for this example.

```
package java11.fundamentals.chapter18;
public class LambdaFunctionalInterfaceExample {

    interface printingSomeText {
        void print(String anyValue);
    }
    public void printLambdaText(String lambdaText, printingSomeText pst) {
        pst.print(lambdaText);
    }
    public static void main(String[] args) {
        LambdaExample lmd1 = new LambdaExample();
         String lambdaText = "Understanding Lambdas";
         printingSomeText pst = (String letsPrint)->{System.out.println(letsPrint);};
         lmd1.printLambdaText(lambdaText, pst);
    }
}
```

The above code produces the following result.

```
Understanding Lambdas
```

As you can observe, all the complexity behind the execution of printing the string has been handled by the interface. The method `printLambdaText` was used in the example only for the execution of the interface's block of code. Let us code the above program further:

```
package java11.fundamentals.chapter18;
public class LambdaFunctionalInterfaceExample2 {
    interface printingSomeText {
        void print(String anyValue);
    }

    public void printLambdaText(String lambdaText, printingSomeText pst) {
        pst.print(lambdaText);
    }
    public static void main(String[] args) {
        LambdaFunctionalInterfaceExample2 lmd1 = new LambdaFunctionalInterfaceExample2();
        String lambdaText = "Understanding Lambdas";

        printingSomeText pst = new printingSomeText() {
            @Override
            public void print(String anyValue) {
                System.out.println(anyValue);
            }
        };

        lmd1.printLambdaText(lambdaText, pst);
    }
}
```

The above program produces the following result.

```
Understanding Lambdas
```

Here, we wrote an implementation for our interface, which was then passed to the `printLamdaText` method. This example was necessary because it can help us realize why we need lambdas. Now, by introducing lambdas, we can recode by writing the following:

```
package java11.fundamentals.chapter18;
public class LambdaFunctionalInterfaceExample3 {

    interface printingSomeText {
        void print(String anyValue);
    }

    public void printLambdaText(String lambdaText, printingSomeText pst) {
        pst.print(lambdaText);
    }

    public static void main(String[] args) {
        LambdaFunctionalInterfaceExample3 lmd1 = new
        LambdaFunctionalInterfaceExample3();
        String lambdaText = "Understanding Lambdas";
        printingSomeText pst = (String letsPrint) -> {
            System.out.println(letsPrint);
        };
        lmd1.printLambdaText(lambdaText, pst);
    }
}
```

The above program produces the following result.

```
Understanding Lambdas
```

Does this not look better? By adding just a single line of a lambda expression, we have improved our code. Lambdas took the parameter and handled its mapping according to our method's parameter list. The code after the arrow can be seen as a concrete implementation of this method.

18.5.2 Examples

Let us code more examples to gain a better understanding of lambda expressions. We will implement the next example with the help of lambda expression, where we have an interface that draws a square.

```
package java11.fundamentals.chapter18;
interface drawSquare {
    public void drawIt();
}
public class LambdaDrawSquareExample {
    public static void main(String[] args) {

        int area = 5;

        // Lambda expression starts. An instance ds of the interface is taken as a parameter.
Note that the absence of the parameters is represented by an empty round bracket
        drawSquare ds = () -> {
            System.out.println("The square is drawn for the area " + area);
        };

        // The concrete implementation provided by the lambda expression draws the square
from the functional interface's method
        ds.drawIt();

    }
}
```

The above program produces the following result.

```
The square is drawn for the area 5
```

18.5.2.1 Lambda Expression without Parameters

We have already mentioned that a lambda parameter can be 0. We demonstrate this with the help of the following example:.

```
package java11.fundamentals.chapter18;
public class LambdaExpressionWithoutParametersExample {
    interface announcement {
        public String announce();
    }

    public static void main(String[] args) {

        // Lambda expression begins
        announcement a = () -> {
            return "The flight going to New York has been cancelled due to the extreme weather";
        };
        // Lambda Expression ends

        System.out.println(a.announce());
    }

}
```

The above program produces the following result.

```
The flight going to New York has been cancelled due to the extreme weather
```

18.5.2.2 Adding a Single Parameter to the Lambda Expression

Now, let us see how we can use a parameter by continuing with our announcement example.

```
package java11.fundamentals.chapter18;

public class LambdaExpressionWithSingleParameterExample {

    interface announcement {
        // Adding a parameter to the method with a String variable
        public String announce(String annText);
    }

    public static void main(String[] args) {
        // First lambda expression begins
        announcement a1 = (annText) -> { // adding a single parameter with optional round
brackets
            return "We have an important announcement to be made. " + annText;

        }; // First lambda expression ends
        System.out.println(a1.announce("The flight going to New York has been cancelled
due to the extreme weather "));
        // Second lambda expression begins
        announcement a2 = annText -> { // adding a single parameter without using round
brackets
            return "We have an important announcement to be made. " + annText;
        };

        // Second lambda expression ends
        System.out.println(a2.announce("The flight going to Boston has been cancelled due
to a hailstorm "));
    }
}
```

The above program produces the following result.

```
We have an important announcement to be made. The flight going to New York has been cancelled due to the extreme weather
We have an important announcement to be made. The flight going to Boston has been cancelled due to a hailstorm
```

18.5.2.3 Lambda Expression with Multiple Parameters

Let us consider that we have to add the total marks of 5 subjects for 5 students. Now, we have an interface for the addition of marks where the method contains the value for subjects. Here, lambda expression can prove beneficial by providing convenience for concrete implementations. Since we have more than a single parameter, we must use round brackets in our expressions. Unlike single parameters, we cannot optionally remove brackets.

```
package java11.fundamentals.chapter18;

public class LambdaExpressionWithMultipleParametersExample {

    interface reportCard {
        // Adding multiple parameters to the method
        public int marksForSubjects(int mathematics,int physics,int biology,int history,
int chemistry);
    }

    public static void main(String[] args) {
        reportCard am1=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry); // round brackets
are used for multiple parameters
        System.out.println(am1.marksForSubjects(74,87,66,53,90));

        reportCard am2=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry);
        System.out.println(am2.marksForSubjects(40,40,50,60,70));

        reportCard am3=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry);
        System.out.println(am3.marksForSubjects(50,60,70,60,70));

        reportCard am4=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry);
        System.out.println(am4.marksForSubjects(64,68,71,67,70));

        reportCard am5=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry);
        System.out.println(am5.marksForSubjects(85,86,55,75,88));
    }
}
```

The above program produces the following result.

```
370
260
310
340
389
```

18.5.3 Using `return`

Now that we have learned how to use various parameters for our use cases, let us move on to the `return` keyword. When a lambda expression contains a single statement, then it is optional to explicitly add or ignore the addition of `return` keyword. However, if there are multiple expressions in the lambda expression, then it is important to add the `return` keyword or you may face an error. Now, going by the above example of adding the marks of students, we can rewrite the program by adding the `return` keyword.

```
package java11.fundamentals.chapter18;
public class LambdaExpressionUsingReturnExample {
    interface reportCard {
        // Adding multiple parameters to the method
        public int marksForSubjects(int mathematics,int physics,int biology,int history,
int chemistry);
    }
    public static void main(String[] args) {
        reportCard am1=(int mathematics,int physics,int biology,int history, int
chemistry)->{
            return (mathematics + physics + biology + history+ chemistry); // adding the
optional return keyword
        };
        System.out.println("The total of the first student is "+am1.
marksForSubjects(74,87,66,53,90));
        reportCard am2=(int mathematics,int physics,int biology,int history, int
chemistry)->{
            return (mathematics + physics + biology + history+ chemistry);
        };
        System.out.println("The total of the second student is "+am2.
marksForSubjects(40,40,50,60,70));
        reportCard am3=(int mathematics,int physics,int biology,int history, int
chemistry)->{
            return (mathematics + physics + biology + history+ chemistry);
        };
        System.out.println("The total of the third student is "+am3.
marksForSubjects(50,60,70,60,70));
reportCard am4=(int mathematics,int physics,int biology,int history, int chemistry)->{
            return (mathematics + physics + biology + history+ chemistry);
        };
        System.out.println("The total of the fourth student is "+am4.
marksForSubjects(65,56,95,65,78));
        reportCard am5=(int mathematics,int physics,int biology,int history, int
chemistry)->{
            return (mathematics + physics + biology + history+ chemistry);
        };
        System.out.println("The total of the fifth student is "+am5.
marksForSubjects(85,86,55,75,88));
    }
}
```

The above program produces the following result.

```
The total of the first student is 370
The total of the second student is 260
The total of the third student is 310
The total of the fourth student is 359
The total of the fifth student is 389
```

Now, let us modify this example by adding multiple statements in each lambda expression. We are adding a statement that can increase the marks of the subject of mathematics by 10 for each student. Without the `return` keyword, these expressions cannot be run.

```
package java11.fundamentals.chapter18;
public class LambdaExpressionWithMultipleParametersExample {
    interface reportCard {
        // Adding multiple parameters to the method
        public int marksForSubjects(int mathematics, int physics, int biology, int
history, int chemistry);
    }
    public static void main(String[] args) {
        // Multiple parameters in lambda expression
        reportCard am1 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry); // mandatory
inclusion of return
        };
        System.out.println("The total of the first student is " + am1.marksForSubjects(74,
87, 66, 53, 90));
        reportCard am2 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry);
        };
        System.out.println("The total of the second student is " +
am2.marksForSubjects(40, 40, 50, 60, 70));
        reportCard am3 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry);
        };
        System.out.println("The total of the third student is " + am3.marksForSubjects(50,
60, 70, 60, 70));
        reportCard am4 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry);
        };
        System.out.println("The total of the fourth student is " +
am4.marksForSubjects(65, 56, 95, 65, 78));
        reportCard am5 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry);
        };
        System.out.println("The total of the fifth student is " + am5.marksForSubjects(85,
86, 55, 75, 88));
    }
}
```

The above program produces the following result.

```
The total of the first student is 380
The total of the second student is 270
The total of the third student is 320
The total of the fourth student is 369
The total of the fifth student is 399
```

18.5.4 Lambdas with Loops

Lambdas are flexible and can also be integrated with loops. Let us consider two students, where each student is assigned the marks of 5 subjects. By using lambdas, we can use the **foreach** loop to add a print statement with each addition.

```
package java11.fundamentals.chapter18;
import java.util.*;
public class LambdaLoopExample {
    public static void main(String[] args) {
        List<Integer> student1 = new ArrayList<Integer>();
        student1.add(50);
        student1.add(60);
        student1.add(70);
        student1.add(80);
        student1.add(90);
        System.out.println("The marks of each subject of student1 :");
        student1.forEach((x) -> System.out.println(x));

        List<Integer> student2 = new ArrayList<Integer>();
        student2.add(90);
        student2.add(80);
        student2.add(70);
        student2.add(60);
        student2.add(50);
        System.out.println("The marks of each subject of student2 :");
        student2.forEach((x) -> System.out.println(x));
    }
};
```

The above program produces the following result.

```
The marks of each subject of student1 :
50
60
70
80
90
The marks of each subject of student2 :
90
80
70
60
50
```

18.5.5 Lambdas with Threads

We can also associate lambdas expressions with threads. Consider you have four threads that have different behaviors. Now, we will code the first thread without lambda expression. Observe how the second thread does the same thing with a better strategy using lambdas.

```
package java11.fundamentals.chapter18;
public class LambdaThreadExample {
    public static void main(String[] args) {
        Runnable run1 = new Runnable() {
            public void run() {
                System.out.println("The first thread is currently in the state of running");
            }
        };
        Thread t1 = new Thread(run1);
        t1.start();

        Runnable run2 = () -> {
            System.out.println("The second thread is currently in the state of running");
        };
        Thread t2 = new Thread(run2);
        t2.start();
        Runnable run3 = () -> {
            System.out.print("The id of the third thread is ");
            System.out.println(Thread.currentThread().getId());
        };
        Thread t3 = new Thread(run3);
        t3.start();

        Runnable run4 = () -> {
            System.out.print("The class of the fourth thread is ");
            System.out.println(Thread.currentThread().getClass());
        };
        Thread t4 = new Thread(run4);
        t4.start();
    }
}
```

The above program produces the following result.

```
The first thread is currently in the state of running
The second thread is currently in the state of running
The id of the third thread is 14
The class of the fourth thread is class java.lang.Thread
```

18.6 | Date and Time API

The recent Java releases have improved the Date and Time API for tackling earlier issues that mainly existed in java.util. Calendar and java.util.Date.

18.6.1 Problems with Earlier APIs

In the past, developers complaint about the following conundrums:

1. Older versions of Calendar and Date APIs were found to be non-safe for threads. What this meant is that Java programmers had to continuously keep adding lines of code. Similarly, there were also issues pertaining to debug concurrency. Thus, it was problematic to debug several segments of the API simultaneously. The newer APIs in Java 9 for Date and Time are a game changer because they have been modified to be safe for threads while they are also plugged with immutability.
2. Older versions of Calendar and Date APIs lacked in design. Their built-in methods were ineffective in running standard tasks for applications. The Date and Time API of Java 9 is said to be ISO centric. All the values pertaining to time,

duration, periods, and date are handled through the use of consistent models. Likewise, standard tasks for daily applications can be performed easily via newer methods.

3. Earlier, time and resources went into the coding of logic related to the time zones of different areas. Newer APIs have automated this task through Local and ZonedDate APIs.

18.6.2 Local Date

The API of LocalDate adheres to the following ISO format:

```
yyyy-MM-dd
```

In the real world, such APIs are used in payroll systems for assigning a payroll to an employee at the beginning or end of the month. Likewise, in applications for storing birthdays such as in social media platforms, there are also requirements for dates. In case you require the latest date, type the following:

```
import java.time.LocalDate;
public class DateExample{
    public static void main(String args[]) {
        LocalDate currentDate = LocalDate.now();
        System.out.println(currentDate);
    }
}
```

The following output will be displayed.

2018-10-12

In order to get the result for any year, month, and day, you can use `of()` method. Similarly, `parse()` method can also be used. Let us see what happens if we use `of()` method:

```
import java.time.LocalDate;
public class DateExample{
    public static void main(String args[]) {
        System.out.println(LocalDate.of(2018, 01, 01));
    }
}
```

By using `parse()` method we can get the same answer:

```
import java.time.LocalDate;
public class DateExample{
    public static void main(String args[]) {
        System.out.println(LocalDate.parse("2018-01-01"));
    }
}
```

In both instances, we get the following answer.

2018-01-01

Likewise, let us go through a few more methods related to dates. Suppose you are developing an application which assigns tasks to employees. Now, an employee can be given a task that has to be done by the following day. In order to add this functionality in the code, you can write the following:

```
import java.time.LocalDate;s
public class DateExample{
    public static void main(String args[]) {
        LocalDate taskDate = LocalDate.now().plusDays(1);
        System.out.println(taskDate);
    }
}
```

As such, you can change the values in the method to get your desired date in future.

Now, suppose you wish to go exactly one month back for your application requirements. This can be done through the `minus()` method.

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
public class DateExample {
    public static void main(String args[]) {
        LocalDate lastMonth = LocalDate.now().minus(1, ChronoUnit.MONTHS);
        System.out.println(lastMonth);
    }
}
```

Consider the following example, where you have to check the date of a specific day. You can parse your date and utilize the method .`getDayOfWeek()` for displaying a date.

```
import java.time.DayOfWeek;
import java.time.LocalDate;
public class DateExample {
    public static void main(String args[]) {
        DayOfWeek whichDay = LocalDate.parse("2018-03-10").getDayOfWeek();
        System.out.println(whichDay);
    }
}
```

Similarly, the day of a month can be returned by using `.getDayOfMonth()` method, as follows:

```
import java.time.DayOfWeek;
import java.time.LocalDate;
public class DateExample {
    public static void main(String args[]) {
        int dayOftheMonth = LocalDate.parse("2018-03-10").getDayOfMonth();
        System.out.println(dayOftheMonth);
    }
}
```

In time related APIs, sometimes circumstances demand the need of identifying a leap year. Leap years can be checked by writing the following code:

```
import java.time.DayOfWeek;
import java.time.LocalDate;
public class DateExample {
    public static void main(String args[]) {
        boolean isItaLeapYear = LocalDate.now().isLeapYear();
        System.out.println(isItaLeapYear);
    }
}
```

For comparing two dates to check when a date occurred in comparison to another date, we can write:

```
import java.time.DayOfWeek;
import java.time.LocalDate;
public class DateExample {
    public static void main(String args[]) {
        boolean beforeOrNot = LocalDate.parse("2018-06-13")
            .isBefore(LocalDate.parse("2018-06-10"));
        System.out.println(beforeOrNot);
        boolean afterOrNot = LocalDate.parse("2018-06-10")
            .isAfter(LocalDate.parse("2018-06-09"));
        System.out.println(afterOrNot);
    }
}
```

18.6.3 `LocalTime`

The `LocalTime` class returns time. It works similar to the `LocalTime` class. For getting the current time, write the following:

```
import java.time.LocalTime;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime whatIsTheTime = LocalTime.now();
        System.out.println(whatIsTheTime);
    }
}
```

To parse time, you can write:

```
import java.time.LocalTime;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime parsingTime = LocalTime.parse("03:20");
        System.out.println(parsingTime);
    }
}
```

The same thing can be done by using `of()` method:

```
import java.time.LocalTime;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime usingOf = LocalTime.of(3,20);
        System.out.println(usingOf);
    }
}
```

To add time, you can use the `.plus()` method. We have used this method to add 5 hours to the current time.

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime fastForward = LocalTime.parse("03:20").plus(5, ChronoUnit.HOURS);
        System.out.println(fastForward);
    }
}
```

To get the hour, we can write:

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
public class TimeExample{
    public static void main(String args[]) {
        int whichHour = LocalTime.parse("03:20").getHour();
        System.out.println(whichHour);
    }
}
```

To compare the time, we can write:

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
public class TimeExample{
    public static void main(String args[]) {
        boolean comparingTime = LocalTime.parse("03:20").isBefore(LocalTime.
parse("02:30"));
        System.out.println(comparingTime);
    }
}
```

Sometimes in DB queries, records are required according to a given time period. To obtain such records, we can get time for noon, minimum, and maximum:

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime maximumTime = LocalTime.MAX;
        System.out.println(maximumTime);
    }
}
```

18.6.4 LocalDateTime

While the above classes are useful for specific cases pertaining to date and time, sometimes both values are required (i.e., a date as well as the exact time for that day). For such scenarios, `LocalDateTime` is used. Now, let us go through its methods. To get the current date and time, we have to write:

```
import java.time.LocalDateTime;
import java.time.LocalTime;
public class TimeExample{
    public static void main(String args[]) {
        System.out.println(LocalDateTime.now());
    }
}
```

To use `of()` method, we write:

```
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class TimeExample{
    public static void main(String args[]) {
        System.out.println(LocalDateTime.of(2018, Month.MARCH, 10, 03, 30));
    }
}
```

The same thing can also be done for parsing:

```
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class TimeExample{
    public static void main(String args[]) {
        System.out.println(LocalDateTime.parse("2018-01-20T06:24:00"));
    }
}
```

For adding and subtracting time, we can use `plus()` and `minus()` methods, just as we have been using them previously.

```
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class DateAndTimeExample {
    public static void main(String args[]) {

        LocalDateTime addHours = LocalDateTime.now().plusHours(3);
            System.out.println(addHours);
    }
}
```

Now, let us see the minus example.

```
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class DateAndTimeExample {
    public static void main(String args[]) {
        LocalDateTime subHours = LocalDateTime.now().minusHours(3);
            System.out.println(subHours);
    }
}
```

Lastly, for getting a specific month, we can simply write:

```
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class DateAndTimeExample {
    public static void main(String args[]) {
        System.out.println(LocalDateTime.now().getMonth());
    }
}
```

18.6.5 `ZonedDateTime` API

In order to combat the issue of time zone, we can use `ZonedDateTime()` in Java 9. In this API, an identifier called ZoneID represents time zones. To create the time zone of any specific city, type the following code in the IDE:

```
import java.time.ZoneId;
public class TimeZoneExample {
    public static void main(String args[]) {
        ZoneId id = ZoneId.of("Asia/Seoul");
        System.out.println(id);
    }
}
```

To get all the time zones that are listed in the API, we can simply write:

```
import java.time.ZoneId;
import java.util.Set;
public class TimeZoneExample {
    public static void main(String args[]) {
        Set<String> allIds = ZoneId.getAvailableZoneIds();
        System.out.println(allIds);
    }
}
```

To choose a specific time zone, we can write:

```
ZonedDateTime specificZone = ZonedDateTime.of(localDateTime, zoneId)
```

Now, let's create a localDateTime:

```
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.Set;
public class TimeZoneExample {
    public static void main(String args[]) {
        LocalDateTime ltd = LocalDateTime.of(2018, Month.MARCH, 10, 07, 20);
        System.out.println(ltd);
    }
}
```

Now, it is possible to add four hours and create a ZoneOffset in the above example. Let us continue the code.

```
import java.time.LocalDateTime;
import java.time.Month;
import java.time.OffsetDateTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;
import java.util.Set;
public class TimeZoneExample {
    public static void main(String args[]) {

        LocalDateTime ltd = LocalDateTime.of(2018, Month.MARCH, 10, 07, 20);
        ZoneOffset os = ZoneOffset.of("+04:00");
        OffsetDateTime osbyfour = OffsetDateTime.of(ltd, os);
        System.out.println(osbyfour);
    }
}
```

The result of localDateTime of method is as follows

2018-03-10T07:20+04:00

Can DateTimeAPI get a user's local date?

Summary

Functional programming is a practice of programming as functions like mathematical functions. Lambda expression is as a way of supporting functional programming in Java. This language is a declarative one, which means logic is expressed without its control flow.

In this chapter, we have learned the following concepts:

1. Functional programming and lambda.
2. Higher order and first order functions.
3. Pure functions and how to use them.
4. Recursion and how to use it in program.

In Chapter 19, we will learn about multithreading and reactive programming. We will learn about the multithreading world and understand the important concepts of concurrency. We will explore various examples and learn about deadlocks, synchronization blocks, lazy initialization, etc.

Multiple-Choice Questions

1. We need to compile Lambda expression to anonymous inner classes.
   (a) True
   (b) False
2. The filter method in Stream API takes in a _____ as argument.
   (a) Predicate
   (b) Consumer
   (c) Function
   (d) Supplier
3. Functional interfaces can have more than one default methods.
   (a) True
   (b) False
4. Which of the following is used to get only the current time?
   (a) `LocalDate.now();`
   (b) `LocalTime.now();`
   (c) `LocalDate.now()`
   (d) All of the above
5. Can lambda expression accept multiple parameters?
   (a) Yes
   (b) No

Review Questions

1. What is lambda?
2. How does functional programming work?
3. What are the benefits of using lambdas?
4. How do we print date in a local format?
5. How do we print date according to time zone?
6. How do we use lambdas with threads?
7. How do we use lambdas with loops?

Exercises

1. Write a program to print stock prices of top 10 companies using lambda and functional programming.
2. Write a program to use DateTime API to print minute-by-minute status of a football match.
3. Create a chart to present all the benefits of using lambdas and functional programming.

Project Idea

Create a chat application using lambda and functional programming. Capture each conversation and use DateTime API to print the conversation as per the user's local time. Consider user's time zone in this case.

Recommended Readings

1. Kishori Sharan. 2018. *Java Language Features: With Modules, Streams, Threads, I/O, and Lambda Expressions*. Apress: New York
2. Pierre-Yves Saumont. 2017. *Functional Programming in Java: How Functional Techniques Improve Your Java Programs*. Dreamtech Press: New Delhi
3. Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft. 2018. *Modern Java in Action: Lambdas, streams, functional and reactive programming*. Manning Publications: New York